Incremental Transformation of Procedural Systems to Object Oriented Platforms

Ying Zou, Kostas Kontogiannis
Dept. of Electrical & Computer Engineering
University of Waterloo
Waterloo, ON, N2L 3G1, Canada
{yzou, kostas}@swen.uwaterloo.ca

Abstract

Over the past years, the reengineering of legacy software systems to object oriented platforms has received significant attention. In this paper, we present a generic re-engineering source code transformation framework to support the incremental migration of such procedural legacy systems to object oriented platforms. First, a source code representation framework that uses a generic domain model for procedural languages allows for the representation of Abstract Syntax Trees as XML documents. Second, a set of transformations allow for the identification of object models in specific parts of the legacy source code. In this way, the migration process is applied incrementally on different parts of the system. A clustering technique is used to decompose a program into a set of smaller components that are suitable for the incremental migration process. Finally, the migration process gradually composes the object models obtained at every stage to generate a amalgamated object model for the whole system. A case study for the migration of a medium size C system to C++ is discussed as a proof of concept.

1. Introduction

Legacy systems are mission critical software systems that entail comprehensive business knowledge and they constitute large assets for organizations. However, their quality and operational life are constantly deteriorating due to the maintenance activities. With the rapid technology updates, there is great pressure to migrate or port existing systems into modern platforms where better and faster operating, development, and maintenance environments exist. One possible solution to leverage the business value of such systems is to re-engineer them into the object-oriented platforms. With properties such as encapsulation, inheritance, and polymorphism inherent in object-oriented designs, the migrated systems can be easier maintained, reused and integrated with other applications in network centric environments.

This work is supported by the IBM Center for Advanced Studies IBM Toronto Laboratory, NSERC, and the Consortium for Software Engineering Research, Canada.

Most of today's legacy systems are written in procedural languages. In a nutshell, the object oriented migration process involves the analysis of the Abstract Syntax Tree (AST) of the procedural code, the identification of object models, and the generation of object oriented code with desired software quality levels. In this context, the software reverse engineering community has proposed methods to migrate systems written in various procedural languages, such as COBOL[6], Fortran[18], and C[12, 13], into object-oriented platforms. In this paper, we propose an incremental source code transformation framework that allows for procedural system to be migrated to modern object oriented platforms. First the system is parsed and a high level model of the source code is extracted. In the proposed framework we introduce the concept of a unified domain model for a variety of procedural languages such as C, Pascal, Cobol, and Fortran. Such unified models can be implemented in XML and denote common language features such as routines, subroutines, function, procedures, types, statements, variables and declarations, just to name a few.
Second, to keep the complexity and the risk of the migration process into manageable levels, a clustering technique allows for the decomposition of large systems into smaller manageable units. A set of source code transformations allows for the identification of an object model from each such unit. Finally, an incremental merging process allows for the amalgamation of the different partial object models into an aggregate composite model for the whole system In this way, the migration task is tackled in a "divide-conquer" manner. The sections below discuss these concepts in detail.

This paper is organized as follows; Section 2 reviews the related work in literature. Section 3 discusses the concepts pertaining to a unified domain model fro the representation of procedural code using XML formats. Section 4 presents an incremental transformation process to migrate procedural systems to object oriented platforms. Section 5 provides a list of transformations and section 6 presents a migration case study. Finally section 7 concludes the paper.

2. Related Work

Source Code Representation

There is a growing stream of activities related to XML representation of source code. In [1], a system for annotating C++ and Java is presented. Specifically, Java and C++ grammars are mapped to corresponding DTDs using domain models. Consequently, semantic actions have been added to custom-made parsers in order to annotate the input stream (source code) with XML tags that are compliant to a domain model DTD. In this approach, common structures between object oriented languages are abstracted in a more generic DTD that aim to model object oriented language constructs.

In [2], an XML based representation of Java source code, called JavaML, is presented. A converter, built with the `Jikes` Java compiler framework, translates from the classical Java source code representation to JavaML, and an XSLT stylesheet converts from JavaML annotated text back into the original source.

In [3], the InterMediate Language (IML) is proposed to model and analyze source code. IML allows for sophisticated data-flow and control-flow analyses to be built. Extensions to IML have been discussed in [13], where the Resource Graph (RG) is proposed to abstract global information, such as call, type, and usage relations for architectural design recovery.

In [4], the Graph Exchange Language (GXL) is proposed as a data exchange format among software analysis tools. GXL is designed for the representation of typed graphs. The CPPX project [5] extracts the C++ facts from the GNU gcc compiler in the binary format and represents the fact into an interchange language for semantic graphs, such as GXL.

Object Oriented Migration

For the migration of procedural code to object oriented designs, there has been significant research activity in systems for migrating COBOL to OO-COBOL[6], Assembly to C[7], C to C++[8, 9]/Java[10], RPG to C++[11]. In the relevant literature, several methods for identifying an object model from a legacy system have also been defined [21 –24]. Overall, these research efforts focus on the identification of objects and abstract data types (ADTs). In [11], the identification of an object model from RPG programs is presented. Objects are centered around persistent data stores, while related parts of code in the legacy system become candidate methods. In [12, 13], an object model is discovered directly from procedural code written in C. Candidate objects are selected by analyzing global data types and function formal parameters. An evidence model helps to attach the
.

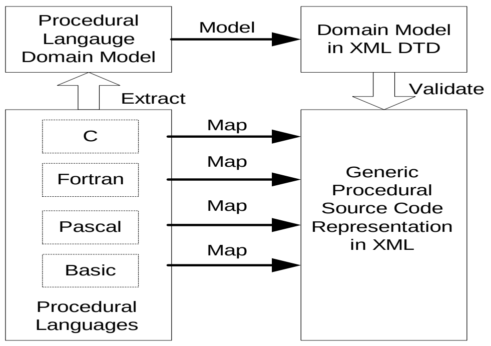


Figure 1: Generic Procedural Code Representation Framework

methods to a candidate class and choose an appropriate object model This evidence model consists of state change information, return types, and data flow patterns. In [14], [20], a concept analysis method is provided to identify modules from C code. It is based on lattice theory to identify similarities among a set of objects based on their attributes. The positive and negative information is used to identify potential modules. Another objectification method is presented in [15]. The method is based on documentation and informal information, such as user manuals, requirement and design specifications, and naming conventions. However, for legacy systems, the external information is not always available, and this technique may not always be applicable. The technique may also be used to analyze source code informal information such as comments and identifier names and from other non-linguistic aspects of OO code. However, there is a need for a systematic approach to control both the complexity of the migration process and the quality of the new migrant system.

3. Source Code Representation

In order to analyze the source code, it is critical to represent the program source code at a higher level of abstraction than source code text. Program representation provides means to generate abstractions, appropriate input to a computational model for analyzing and reasoning about programs, and methods for the translation and normalization of programs. In this section, we discuss the program representation techniques that are based on procedural language domain models and the XML markup language. To build a generic representation for specific categories of procedural languages, there are two major steps involved, namely, the abstraction of the individual procedural language domain models and the representation of such an abstraction in a generic format, as illustrated in Figure 1. The focal point is to identify the functional equivalent constructs and aggregate them at a higher level of abstraction

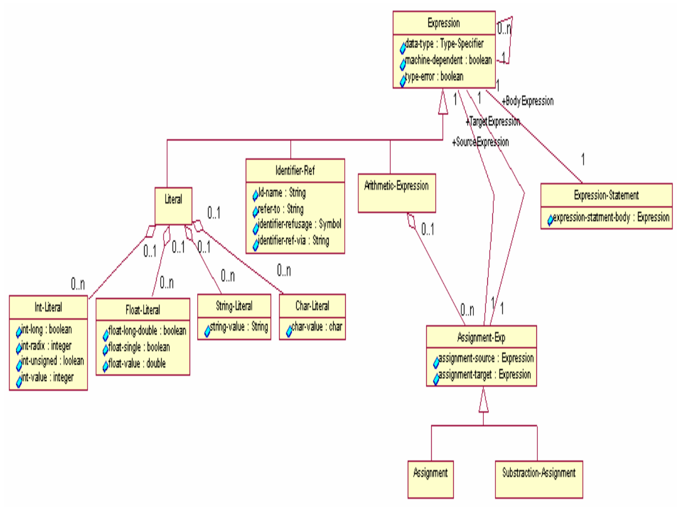


Figure 2: Sample Domain Model for Expressions

```
<EXPRESSION-STATEMENT >
  <EXPRESSION-STATEMENT-BODY>
    <EXPRESSION>
      <ASSIGNMENT-EXP>
        <ASSIGNMENT surface-syntax="shuffle_level = num_decks * 26">
          <ASSIGNMENT-TARGET surface-syntax="shuffle_level">
            <IDENTIFIER-REF id-name="shuffle_level"/>
          </ASSIGNMENT-TARGET>
          <ASSIGNMENT-SOURCE surface-syntax="num_decks * 26">
            <MULTIPLICATION surface-syntax="num_decks * 26">
                <MULTIPLICATION-ARGS>
                   <IDENTIFIER-REF id-name="num_decks"/>
                   <INT-LITERAL  int-long="NIL" int-radix="10"
                                int-unsigned="NIL" int-value="26"/>
               </MULTIPLICATION-ARGS>
            </MULTIPLICATION>
          </ASSIGNMENT-SOURCE>
        </ASSIGNMENT>
    </ASSIGNMENT-EXP>
  </EXPRESSION>
 </EXPRESSION-STATEMENT-BODY>
</EXPRESSION-STATEMENT>
```

Figure 3: XML Element Structure for Expression Statement in C

For example, language constructs such as “subroutine” in Fortran and “Function” in C, denote similar concepts that for the purpose of language modeling can be aggregated by a unique term “procedure”. Due to size limitations in this paper we will focus our discussion on a domain model that stems mostly from the C programming language. The specific domain model is generic enough to handle constructs from various programming procedural languages such as Fortran, Pascal, and Cobol. A subset of such language domain model for Expressions represented in UML is illustrated in Figure 2. For this purpose, the UML object-oriented modeling language is utilized to denote language syntactic constructs and AST edges as associations. The association links represent the attributes of non-primitive types and are denoted as mappings from one class to another. As shown in Figure 2, `expression` is a basic construct, including sub-classes, such as `identifier reference`, `literal constant`, `arithmetic expression`, or `assignment expression`. The `expression`, as a base class, contains the common attributes and shares them with its subclasses, such as `literal`, `arithmetic expression`, `identifier reference` by inheritance. The `expression` is self-reflective, with which an association points back to itself. Therefore, an `expression` can contain one or more `expressions`.

3.1. Source Code Representation Using XML

3.1.1. Representation of ASTs in XML

There are two approaches to extract the abstract syntax tree and encode it in XML. The bottom-up approach, utilizes the concept of a domain model definition that denotes the syntactic structures of a programming language such as Pascal, Fortran, and C. Tools that utilize this approach include Refine for C/Fortran/COBAL, Datrix for C++/C/Java.

The other approach, referred to as the top-down approach, examines the grammar of the specific programming language, and defines a standard logical structure for an annotated Abstract Syntax Tree. By following the language grammar rules, different parsers can extract the necessary information from the source code and encode it in a uniform and language-neutral format. Using a domain model definition extracted from the specification of a given programming language (i.e. ANSI C), a hybrid approach to define the logical structure of the entities of an Abstract Syntax Tree in terms of a Document Type Definition (DTD) document can be utilized by following the steps below.

In the first step, a domain model for a given programming language is defined as a collection of classes, hierarchies, and association. By recursively traversing the hierarchy of the domain model entities, the given domain model can be mapped to a Document Type Definition (DTD). Specifically, domain model classes are mapped as DTD elements, and associations are mapped as DTD attributes. During parsing the semantic actions of the parser can be used to generate an XML representation of the source code as illustrated in Figure 3. In the second step, the domain model for a given language and its corresponding DTD can be enhanced with information

Fortran Domain Model	***Generalized Domain Model***
Structure-Statement	Structure
Record-Statement	Struct Variable Declaration
Common-Statement	Global Variable Declaration
Programs	Program
Executive Program	File
Program Unit	Function-Def
Type-Statement	Declaration
Read-Statement	Function-Call
Call-Statement	Function-Call
Indexable-Name/function-Params-	Function-call
Character-Statement	String
Equivalence-Statement	Union-Struct
Intrinsic-Statement	Function-Pointer

Table 1: Generalization of the Fortran Domain Model.

such as unique identifier numbers, linkage, and analysis information. Similarly, domain model generalizations include the introduction of elements that relate to system constructs such as system, module, and component.

In this context, the generic XML based representation for procedural code can be designed as to contain common language structures found in a group of programming languages including files, libraries, data types, data definitions, variables, constants, macros, expressions, statements, I/O utilities and functions.

For our work, the AST of each individual procedural language is extracted and represented in the XML format. The XSLT (eXtensible Stylesheet Language transformation) is used to define transformation rules to generalize individual domain models represented as a DTD to more generic domain models. An example mapping for Fortran constructs is illustrated in Table 1.

4. Incremental Migration Process

In this section, an incremental process model to migrate legacy systems into object-oriented platforms is presented. The need for an incremental migration process is strong since large systems are not migrated at once because of the complexity and the risk involved. It is therefore important that a technique that identifies system segments that serve as “work areas” in the migration process to be devised. We aim to divide a system into a collection of cohesive “work areas”, which group exclusively related entities for the migration process. Consequently, the XML based AST for every such “work-area” segment is generated, and in turn the migration process operates iteratively on each segment (shown in Figure 4).

In the following subsections, these issues are addressed in detail.

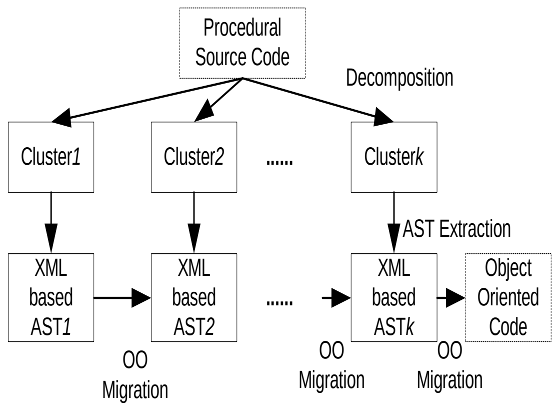


Figure 4: Incremental Object Oriented Migration Process

4.1. A System Segmentation Algorithm

Most clustering techniques presented in literature utilize certain criteria to decompose a system into a set of meaningful modular clusters. Such criteria attempt to achieve a cluster with low coupling, high cohesion, interface minimization and shared neighbors. In the context of the object-oriented migration, we strive to produce clusters that assemble the maximum source code properties related to a class candidate. In this respect, essential source code entities are called *seeds*. Other entities that *associate* with this seed entity form a cluster. An *association* is a directed edge from a *seed* to its related *entities*.

Criteria on the Selection of a *Seed*
A *seed* is selected according to its potential to be considered a class candidate in the new migrant system. In this context, a *seed* can be chosen from aggregate data types, global variable declarations, function parameter declarations, and function pointer declarations. Specifically, the aggregate data types include *struct* type definitions, *union* type definitions, *arrays*, and *enumeration* definitions. In this case, the fields in an aggregate data type become the data members in a class candidate. Similarly, a global variable is encapsulated as a data member in a class candidate. Moreover, a function pointer declaration is treated as a clustering *seed* for the reason that a function pointer declaration defines a type for the functions passed as parameters.

Criteria on the Selection of *Entities*
Due to the object oriented design principle that a class encapsulates data and the related methods, we focus on the discovery relations between data declarations and functions that use such data. Such relations include type references, data updates, and data uses. The algorithm for selection of *entities* given a *seed* is illustrated in Program 1. Furthermore, the clustering algorithm, illustrated in

Program 2, is composed of three major steps. First, all the functions in the original procedural system are identified and stored in a set F. Second, all the *seed* candidates are identified and stored in a sequence. Finally, for each *seed* the associated *entities*, including functions and aggregated data types are collected in a cluster. Consequently, every result cluster is represented as a tuple in the form of a tuple *<seed, associated function set, associated aggregated data type set>*.

The overall segmentation algorithm takes the AST(Abstract Syntax Tree) as an input, and produces clusters represented by a sequence of tuples as an output. The pseudo code programs below illustrate the process for identifying migration work-area segments.

Program 1: Algorithm for Collecting Related Entities

```
Collect_Related_Entities(T_i, F, T, S)
Input:
    T_i: an aggregate type, a global variable, or function pointer type considered to be a seed
    F: a set of all functions
    T: a set of all aggregated types and global variable declarations
    S: AST view of a system
Output:
    P : a tuple contains T_i, a set of related functions and a set of related types
Algorithm:
Begin:
    --Initialize the set of related functions
    $M_{T_i} = \varnothing$;
    --Initialize the set of related aggregated
    $R_{T_i} = \varnothing$;
    -- T_j has data member in the type of T_i
    $R_{included\_types} = \{T_j \mid T_j \in T \text{ and } T_i \in T \text{ and } isType(hasDataMember(T_j)) = T_i\}$;
    -- T_j is cased into the type of T_i
    $R_{casted\_types} = \{T_j \mid T_j \in T \text{ and } T_i \in T \text{ and } isCastedTo(T_j) = T_i\}$;
    --T_j and T_i have data members
    $T_{cloned\_types} = \{T_j \mid T_j \in T \text{ and } T_i \in T \text{ and } hasDataMembers(T_j) \subseteq getDataMembers(T_i)\}$;
    -- T_j and T_i have data members that are mapping to each other
    $T_{mapped\_types} = \{T_j \mid T_j \in T \text{ and } T_i \in T \text{ and } hasMappedTo(hasDataMember(T_j)) = hasDataMember(T_i)\}$;
    $R_{T_i} = T_{included\_types} \cup T_{casted\_types} \cup T_{cloned\_types} \cup T_{mapped\_types}$;
    -- F_j has parameters with the type of T_i
    $M_{parameter} = \{F_j \mid F_j \in F \text{ and } isType(hasParameter(F_j)) = T_i\}$
    -- F_j has return value with the type of T_i
    $M_{return} = \{F_j \mid F_j \in F \text{ and } isType(hasReturn(F_j)) = T_i\}$
    -- F_j update variables related to T_i
    $M_{update} = \{F_j \mid F_j \in F \text{ and } T_i \subseteq updatedTypes(F_j)\}$
    -- Fj is actual passing function to the function pointer type of
    -- parameter
    $M_{actual\_functions} = \{F_j \mid F_j \in F \text{ and } isFunctionPointerDeclaration(T_i) \text{ and } getType(F_j) = T_i\}$
    $M_{T_i} = M_{parameter} \cup M_{return} \cup M_{update} \cup M_{actual\_functions}$
    P←<Ti, $M_{T_i}$, $R_{T_i}$>;
    return P
End
```

Program 2: Algorithm for Clustering

```
Segment_System (S)
Input:
    S: AST view of a system
Output:
    S_p: set of clusters
Algorithm:
Begin
    -- Initialize a partition P into an empty tuple
    P= ∅;
    -- Initialize the cluster set S_p into an empty set
    S_p=∅;
    -- Identify all the functions and store them in a set F
    F ← getAllFunctions(S);
    --Identify all seed candidates
    [T_1, T_2, …, T_n] ← getAllSeeds(S);
    T ← [T_1, T_2, …, T_n];
    -- Identify clusters
    for each type T_i in T loop
        $< T_i, \{F^1_{T_i}, ..., F^k_{T_i}\}, \{T^1_{T_1}, ..., T^m_{T_i}\} > \leftarrow$
                                        Collect_Related_Entities(T_i);
        $P \leftarrow < T_i, \{F^1_{T_i}, ..., F^k_{T_i}\}, \{T^1_{T_1}, ..., T^m_{T_i}\} >$;
        S_p ← S_p with P;
    end loop;
    S_p = [P_1, P_2, …P_n];
    return S_p
End
```

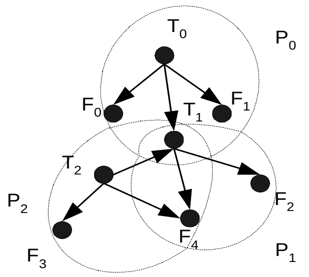


Figure 5: Example for System Segmentation Algorithm

Figure 5, illustrates a result of applying the clustering algorithm, where three clusters have been identified:

$P_0 = <T_0, \{F_0, F_1\}, \{T_1\}>$,
$P_1 = <T_1, \{F_2, F_4\}, \emptyset>$,
$P_2 = <T_2, \{F_3, F_4\}, \{T_1\}>$.

The algorithm and the clustering criteria allow for overlapping regions to exist. When the overlap occurs on aggregate data types, it may indicates a multi-inheritance relationship among the generated class candidates. However, if an overlap occurs on functions it reflects conflicts in method assignment. As one function can be only attached to one class as a method, the conflict has to be resolved. In order to achieve good quality in the migrated system, we provide a qualitative method and an evidence model to determine the choice of the class candidate that the function should be attached to [16, 17]. Finally, in the context of incremental migration process, it is important to independently select and migrate a cluster without relying on the information in other clusters. In this respect, the shared regions are duplicated in each cluster while converting the clusters in the XML format. In other cases that some functions are not related to any seeds, we wrap such "leftover" functions into one cluster.

4.2. Incremental Migration Process

The decomposition of a program produces a set of smaller work areas. The algorithm for the incremental transformation is illustrated in Program 3. It takes the sequence of identified clusters as input and generates incrementally an object-oriented system in the end. The migration process is divided into *k* phrases one for each cluster. The algorithm iterates over each cluster, and updates the system object model, referred to as OM. It is worth to note that the clusters are applied in the order that is constrained by the function calls inside the shared functions. As previously stated, the shared functions will cause the conflicts in method assignment. The conflicted functions called by other functions in conflicts should be resolved first. Therefore, the clusters with less function call dependencies in the shared functions are migrated first. In summary, the transformations are performed in four steps, (as shown in Program 4):

These steps are:

1) generate new class candidate which is added into the object model;
2) attach the associated functions into the new class candidate, and update the object model;
3) resolve conflicts when a function can be assigned to either the current class candidate or the existing class candidate in the object model (the resolving techniques have been presented in the research papers [16, 17]); and finally,
4) refine the object model by identifying class associations.

```
Transform_Clusters($S_p$)
Input:
  -- a sequence of clusters, k is the number of the clusters
  $S_p=[P_1,P_2,\ldots,P_k]$
Output:
   OM: object model of the system
Algorithm:
Begin:
   OM=$\emptyset$;
   Order_Clusters($S_p$);
    while (phrase < k) do
      $OM_{phrase}$=Generate_Object_Model(OM, $P_{phrase}$);
      OM= $OM_{phrase}$;
      phrase = phrase + 1;
   end while
   return OM
End
```

Program 4: Algorithm for Generating Object Model from a Cluster

```
Generate_Object_Model($OM_{i-1}$, $P_i$)
Input:
   $OM_{i-1}$: the accumulated object model from clusters
   1..i-1;
   $P_i$: the ith cluster;
Output:
  OM: the accumulated object model from clusters 1..i;
Algorithm:
Begin:
   OM=Generate_Class($OM_{i-1}$, $P_i$);
   OM=Attach_Methods(OM, $P_i$);
   OM=Resolve_Conflicts(OM, $P_i$);
   OM=Refine_Object_Model(OM);
   return OM;
End
```

The transformations that allow for generating class candidates, attach methods to classes, resolving conflicts, and refine the obtained object model are outlined in the following sections.

5. Object Model Identification

This part of the process is divided into three steps namely, class identification, private data member identification and method attachment. The following sections provide indicative transformations that can be applied in each step.

5.1 Class Identification

The first step towards the migration of procedural source code to an object-oriented platform is the selection of possible object classes. This task can be automated to a large extend using a number of different software analysis techniques. However, no matter how sophisticated the analysis techniques are, user assistance and guidance is crucial on obtaining a viable and efficient object model.
Significant domain information can be utilized by the user to guide the discovery process and to obtain a better and more suitable object model. The object identification techniques focus on two areas: a) the analysis of global variables and their data types, b) the analysis of complex data types in formal parameter lists. Analysis of global variables and their corresponding data types is focusing on the identification of variables that are globally visible within a module. For each variable its corresponding type is extracted from the Abstract Syntax Tree, and a candidate object class is generated. Data type analysis is focusing on type definitions that are accessible via libraries. Examples include `typedef` C constructs. Data types that are used in formal parameter lists become also primary class candidates. The union of data types that are identified by the global variable analysis and data type analysis forms the initial pool of candidate classes.

5.2 Private Data Member Identification

Data type analysis
Aggregate data types refer to a collection of data members inside a user-defined source code structure, such as `struct` and `union` in C. The pre-condition of this transformation rule requires that the `struct` type is not defined inside any other `struct` type. Since such `struct` type is globally available to be referenced by functions and can be used by other declarations throughout the program, it is will be suitable to be a class candidate in the new system. The post-condition characterizes the result of the transformation that all of the data members of the `struct` type become the private class attributes. Similarly, the `union` type can be converted into class candidate with the pre-condition that it is not embedded inside any other `struct` type definitions.

Variable analysis
Although C++ allows for global constant definitions to be accessible within file and global scope, keeping these scopes of variables unchanged in the new system would violate the principles of encapsulation and information hiding in the target object oriented system. The transformation rule aims at eliminating such extensive scopes, by encapsulating such declarations as a private data member in an individual class.

5.3 Method Attachment

Parameter type analysis
A formal parameter in a procedure or a function indicates that the function references a data item of a particular type. In the process of object model extraction, we consider procedures and functions as method candidates. To maximize the cohesion inside the class and minimize the coupling between classes, the procedures and the function with `struct` parameter types are attached to the class candidates that are generated from these `struct` types.

Return type analysis
The return type of a function indicates that the function possibly uses and/or updates the data fields of the aggregate type of the return value. Especially, in the case that a function without a parameter of an aggregate type, the return type provides strong evidence to assign such a function to the class candidate originated from the return type.

Variable usage analysis
In the case that a function has neither aggregate type parameters, nor a return value of a aggregate type, the frequency of usage of aggregate types in the function body is considered as an evidence to transform the function to method in the class candidate that is generated by the aggregate type used.

6. Object Model Refinement

6.1 Inheritance Identification

Data type cast
In cast operations, the compiler will automatically change one type of data into another when appropriate. Casting allows to make this type conversion explicit, or to force it when it wouldn't normally happen. Implicit cast operation between two data types suggests that these data types share common data fields or are interchangeable.

Struct in struct
The data fields of a `struct` type are a group of variable declarations, the type of which can be another `struct` type. In this context, the outer `struct` type reuses the

definition of the inner `struct` type. This feature can be said that it implies the inheritance relation between the class candidates generated from the inner `struct` type and the outer `struct` type.

`Union` type defined in a `struct` type

`Union` types denote that their data members share the same memory space. This source code feature of the original procedural system provides a subtle difference from the semantics of a C `struct` where all members are referenced as a single distinct group. By contrast, here only one `union` data member can be referenced at a time, and different data members cannot co-exist in the same time. In the case that union type is defined in the scope of another structure definition, the common structure of the union data member and the rest of the `struct` data fields can be extracted as a super-class, while each of the union data members can be transformed to a subclass.

Data clones

If two or more structures differ only with respect to few fields, the common fields of these structures can be extracted in order to form a super class. Moreover, subclasses can inherit from it with their non-common fields as their private attributes.

6.2 Polymorphism Identification

Switch statement replacement

One of the most important characteristics of quality source code designs is the limited use of `switch` (or `case`) statements. A `switch` statement in the procedural code that uses in its evaluation condition expression a type code, can be replaced by a set of polymorphic methods. The type code that is used to determine which `case` statement will be invoked can be transformed to an individual class with an abstract polymorphic method. Furthermore, each of the possible values of the type code may form sub-classes, and define a concrete polymorphic method that corresponds to case statement bodies labeled by the type code value. The same transformation rule for generating inheritance relations applies also when a function has a type code as its parameter to indicate that its behavior is determined by the value of the parameter.

Function pointer replacement

There are two ways C functions can be invoked namely, by name and by address. Invocation by name is by far the most common one when the functions to be called are decided at the compile time. Invocation by address is used to determine at run time the concrete functions to be executed. In this context, each possible function pointer reference can become a class and their corresponding source code can become a polymorphic method.

A comprehensive set of transformation rules to perform each of above migration steps are presented in [16], [17]. To govern the order of transformation composition, the pre/post conditions for each transformation are formally specified in OCL.

7. Case Studies

Source Code Representation

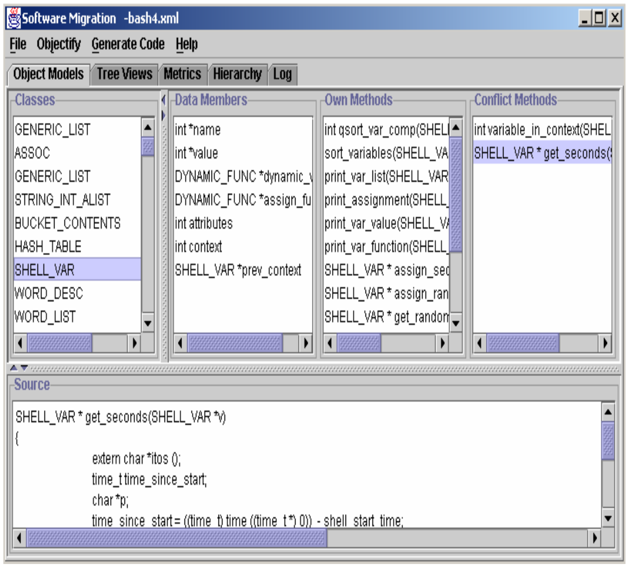
To investigate the effectiveness of the generic procedural source code representation framework, the domain model for the C programming language was examined and generalized. Furthermore, the C source code for various systems has been represented in the form of XML DOM trees. We have used the Refine/C Parser by Reasoning to obtain an XML version of the C source code. In this context, we could have used any parser for this task, but we have chosen the Refine parser because of the flexibility of its API. Table 2 provides some comparison statistics related to the size of the original source codes, the number of clusters identified and the average size of a cluster in XML format. As shown in the Table 2, each component is of a manageable size for the software analysis purposes.

Object Oriented Migration

For our experiments, we have applied the proposed incremental migration technique to extract an object model for the BASH (Bourn Again SHell) originally written in C. A software analysis tool that us based on the proposed migration process was developed to migrate it into C++. For this case study, we have identified 186 classes including 91 classes generated from the aggregate data types as seeds and 95 classes from global variable declarations as seeds. An example object model generated in the middle of the migration process is illustrated in Figure 6. The highlighted part in Figure 6 illustrates *SHELL_VAR* as a newly identified class with two possible methods (*variable_in_context* and *get_seconds*). These methods are in conflict with the class candidate since can also be assigned to other class candidates as well. For each of the methods in conflict, the choice to which class to assign the method is determined by the quality impact on specific software qualities. The method is attached to the class with a higher likelihood to achieve the high cohesion and low coupling [25], [26], [27]. The detailed explanation to the likelihood computation is presented in [16, 17]. The tables on the right side of the screenshot (as shown in Figure 7) provides the quality computation result for method *get_seconds* with the respect of cohesion and coupling if the method is assigned into either *SHELL_VAR* or *seconds_value_ assigned*.

System	***C Source Code Size***	***Cluster #***	***System Size in XML***	***Avg. Cluster Size in XML***
AVL Tree	168,286 Bytes	9	1.69MB	278,722 Bytes
CLIPS	983,127 Bytes	325	38.03MB	423,084 Bytes
BASH	1,257,838 Bytes	327	16.9MB	404,992 Bytes

Table 2: System Segmentation Result



GENERIC_LIST
ASSOC
GENERIC_LIST
STRING_INT_ALIST
BUCKET_CONTENTS
HASH_TABLE
SHELL_VAR
WORD_DESC
WORD_LIST

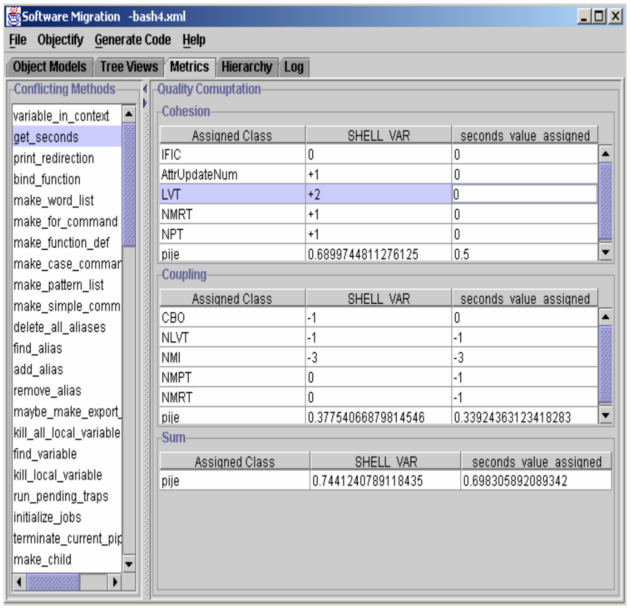
int *name
int *value
DYNAMIC_FUNC *dynamic_
DYNAMIC_FUNC *assign_fu
int attributes
int context
SHELL_VAR *prev_context

int qsort_var_comp(SHEL
sort_variables(SHELL_VA
print_var_list(SHELL_VAR
print_assignment(SHELL_
print_var_value(SHELL_V
print_var_function(SHELL_
SHELL_VAR * assign_sec
SHELL_VAR * assign_ran
SHELL_VAR * get_randon

int variable_in_context(SHEL
SHELL_VAR * get_seconds(

```
SHELL_VAR * get_seconds(SHELL_VAR *v)
{
                extern char *itos ();
                time_t time_since_start;
                char *p;
                time_since_start = ((time_t) time ((time_t *) 0)) - shell_start_time;
```

Figure 6: Example Class Breakdown of BASH System



variable_in_context
get_seconds
print_redirection
bind_function
make_word_list
make_for_command
make_function_def
make_case_commar
make_pattern_list
make_simple_comm
delete_all_aliases
find_alias
add_alias
remove_alias
maybe_make_export_
kill_all_local_variable
find_variable
kill_local_variable
run_pending_traps
initialize_jobs
terminate_current_pip
make_child

Assigned Class	SHELL_VAR	seconds_value_assigned
IFIC	0	0
AttrUpdateNum	+1	0
LVT	+2	0
NMRT	+1	0
NPT	+1	0
pije	0.6899744811276125	0.5

Assigned Class	SHELL_VAR	seconds_value_assigned
CBO	-1	0
NLVT	-1	-1
NMI	-3	-3
NMPT	0	-1
NMRT	0	-1
pije	0.37754066879814546	0.33924363123418283

Assigned Class	SHELL_VAR	seconds_value_assigned
pije	0.7441240789118435	0.698305892089342

Figure 7: Quality Computation for Method in Conflicts

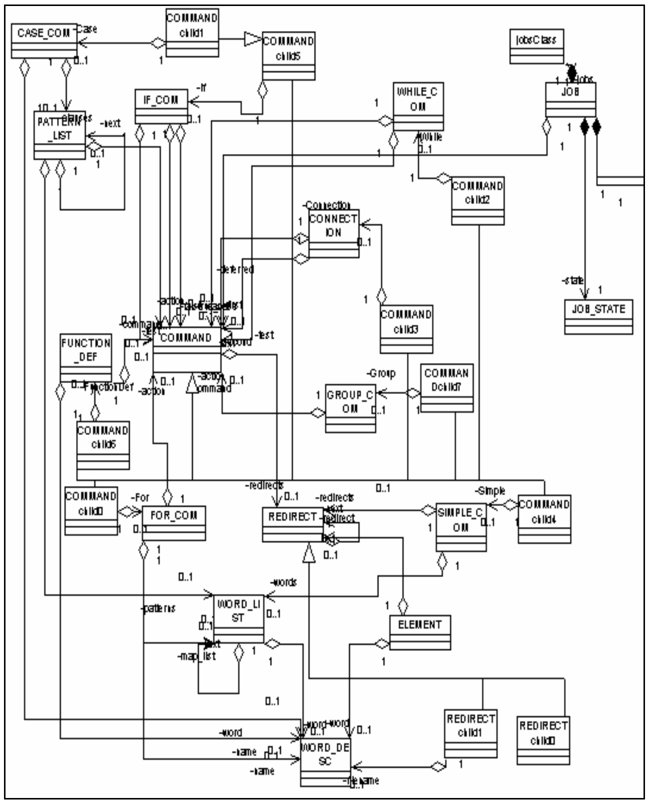


Figure 8: Class Diagram of Migrated System

The co-efficient Pije refers to the likelihood to achieve low coupling and high cohesion. The third table summates the transformation effects on these two quality factors. As a result, the assignment to *SHELL_VAR* gives higher contributions. In addition, the first column in the Figure 7
lists the order of resolving the functions in conflicts. Finally, a partial of the class diagram for the migrated system is illustrated in Figure 8.

8. Conclusion

In the context of the object-oriented migration, a generic reengineering framework should consist of four key elements, that include a unified model for source code representation, incremental transformation process for the migration of large systems, a comprehensive set of transformation rules, and a quality control mechanism to ensure the migrated system with the desired software quality.

In this paper, a unified source code representation framework that utilizes language domain models is represented in XML and Data Type Definition documents. The focal point is to select a model that is rich enough to express all the possible syntactic constructs in the procedural languages. To facilitate the analysis of large systems a segmentation algorithm is provided to decompose a program into a set of smaller components where incremental migration can be achieved. In such a way, a large system can be reengineered gradually in order to reduce the risk and computation costs involved. Finally, results from a case study to identify an object model for the Bash Unix shell are presented.

Future work will include the design of wrappers that allow for the integration of system components that have been already migrated to an object oriented platform with the rest of the legacy system that is still is in its original procedural form.

References

[1] E. Mamas, K. Kontogiannis, "Towards Portable Source Code Representation Using XML", 7th WCRE'2000, November 2000.

[2] Greg Badros, "JavaML: A Markup Language for Java Source Code", http://www.cs.washington.edu/homes/gjb/papers/javaml/javaml.html.

[3] R. Koschke, J.-F. Girard, and M. Würthner, "An intermediate representation for integrating reverse engineering analyses" , 5th WCRE'2000, November 2000.

[4] R. C. Holt, et al., "GXL: Toward a Standard Exchange Format", 7th WCRE 2000, November 2000.

[5] http://swag.uwaterloo.ca/~dean/cppx/

[6] Harry Sneed, "Object Oriented COBOL Recycling", in the Proceedings of 3rd Working Conference of Reverse Engineering, 1996.

[7] M. P. Ward, "Assembler to C Migration using the FermaT Transformation System",

[8] Kostas Kontogiannis, et. al. "Code Migration Through Transformations: An Experience Report", In the Proceedings of CASCON 1995.

[9] Michael Siff and Thomas Reps, "Identifying Modules via Concept Analysis", IEEE Transactions on Software Engineering, Vol. 25, No. 6, Nov. 1999.

[10] Johannes Martin and Hausi A. Muller, "C to Java Migration Experiences", in the Proceedings of the Sixth European Conference on Software Maintenance and Reengineering March 2002.

[11] De Lucia, G.A. Di Lucca, A.R. Fasolino, P. Guerra, S. Petruzzelli, "Migrating Legacy Systems toward Object-Oriented Platforms", 1997, IEEE.

[12] Ying Zou, Kostas Kontogiannis, "A Framework for Migrating Procedural Code to Object Oriented Platform", in the proceedings of 8th Asia-Pacific Software Engineering Conference, 2001.

[13] K. Kontogiannis, P. Patil, "Evidence Driven Object Identification in Procedural Systems'', STEP'99, September 1999, pp. 12-21.

[14] Michael Siff and Thomas Reps, "Identifying Modules via Concept Analysis", IEEE Transactions on Software Engineering, Vol. 25, No. 6, Nov. 1999.

[15] Letha H. Etzkorn, Carl G. Davis, "Automatically Identifying Reusable OO Legacy Code", Computer, IEEE, October, 1997.

[16] Ying Zou, Kostas Kontogiannis, "Quality Driven Transformation Compositions for Object Oriented Migration", the 9th IEEE Asia Pacific Software Engineering Conference (APSEC), Gold Cost, Queensland, Australia, December 2002 , pp. 346-355.

[17] Ying Zou, Kostas Kontogiannis, "Migration to Object Oriented Platforms: A State Transformation Approach", the 19th IEEE International Conference on Software Maintenance (ICSM), Montreal, Quebec, Canada, October 2002, pp.530-539.

[18] Theurich, G.; Anson, B.; Hill, N.A.; Hill, *A.;* "Making the Fortran-to-C transition: how painful is it really?" Computing in Science & Engineering, Volume: 3 Issue:1, Jan/Feb 2001 Page(s): 21 –27

[19] Martin Fowler, "Refactoring: Improving the Design of Existing Code", Addison-Wesley, 2000.

[20] C. Lindig and G. Snelting, "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis", In Proc. Of International Conference on Software Engineering, 1997.

[21] H. A. Sahraoui, W. Melo, H. Lounis, F. Dumont, "Applying Concept Formation Methods To Object Identification In Procedural Code", In Proc. Of 12th Conference on Auotmated Software Engineering, 1997.

[22] Letha H. Etzkorn, Carl G. Davis, "Automatically Identifying Reusable OO Legacy Code", Computer, IEEE, October, 1997.

[23] Aniello Cimitile, et.al, "Identifying Objects In Legacy Systesm Using Design Metrics", The Journal of Systems and Software 44 (1999), Elsevier.

[24] De Lucia, G.A. Di Lucca, A.R. Fasolino, P. Guerra, S. Petruzzelli, "Migrating Legacy Systems toward Object-Oriented Platforms", 1997, IEEE.

[25] Stephen H. Han, "Metrics and Models in Software Quality Engineering", Addison-Wesley, 1995.

[26] S.R. Chidamber, C.F. Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transaction, Software Engineering, 1994.

[27] W. Li, and S. Henry, "Object-Oriented Metrics Which Predict Maintainability", Journal of Systems Software, 1993.